A Buddy System memory allocator (with bitmap)

Benefits

This memory allocator manages a pool of storage. It is written in C. The user initialises it in two calls. First they hand it a pool of storage to manage, a bitmap containing one bit for every block in the pool, and an array big enough to hold one size_t for every possible block size (normally <= 32). Then they explicitly release regions of store within the pool (normally the entire pool). After this they can use an interface vaguely reminiscent of malloc(), free(), and realloc().

Every block of storage returned by poolMalloc() and poolRealloc() actually contains 2^n bytes, for some n, and starts on an offset from the start of the pool that is a multiple of 2^n. No 'block headers' are needed for allocated blocks of memory (the bitmap holds all the information needed). This means that it is possible to allocate every byte in the pool to the user, as one large block, or as whatever collection of power-of-two sized blocks happens to fill up the pool. This allocator never takes very long to allocate, reallocate, or free storage: the time taken goes up as the number of different block sizes, (which will be <= 32 on most machines, since each block size is a power of two), plus the size of the block being allocated or deallocated. A cleverer and more fragile version of this idea could guarantee that the time taken would go up only with the number of different block sizes. This code has been written to include as many consistency checks as possible, while still retaining the predictable worst case time bounds mentioned above. This is partly to make it pick up user code that overwrites the areas allocated to it, and partly because it is still being debugged itself!

Enough of the 'pointers' used in this code are actually block numbers that it should be possible to use this code to manage a block of shared memory which might appear at different offsets to different processes. I haven't tested this, and this code doesn't include the necessary synchronisation calls for this to work. You would also have to think very hard about problems like one process dying and leaving these data structures inconsistent.

The "Buddy System" algorithm used is a variant of that described as Algorithm R of section 2.5 of Volume 1 of Knuth's "The Art of Computer Programming", with note taken of exercises 25 and 29 of that section.

Drawbacks

The dark side of this allocator is that (except for the test code I provide) it has never been used, so it almost certainly still has bugs. I have tested it only on Redhat LINUX and Borland C++ 4.52. On Borland you get scores of compiler warnings. These are either because I haven't bothered to find the right windows header files, or because it thinks some of my stranger casts (inevitable inside a storage allocator) are suspicious. The test program still seems to work. These programs were really only written to check that the

idea could be made to work. Also, the buddy system itself is wasteful of storage unless you really on require blocks whose size is a power of two, and which must be aligned on a boundary that is a multiple of the block size.

You should really use..

Thanks to Niall Murphy (home page at http://www.iol.ie/~nmurphy), I can point you to a paper which describes a large variety of memory allocators, and gives figures for the memory fragmentation of the buddy system in practice. The paper is "The Memory Fragmentation Problem: Solved?", by Mark S. Johnstone and Paul R. Wilson. It lives near www.cs.utexas.edu/users/oops/papers.html. This paper refers to a *very much better allocator than this*, by Doug Lea. This can be found near http://g.oswego.edu. Another link, referenced by the Microsoft paper, "Memory Allocation for Long-Running Server Applications", is http://www.cs.colorado.edu/~zorn/Malloc.html

Location

The code and header file of this *pre-beta quality prototype* are in pool.c and pool.h. Its *only* test, demonstration, or user application, is pool_test.c.